

Going straight with PID

How to make your Raspberry Pi robot drive in a straight line



Martin O'Hanlon

@martinohanlon

Martin is the co-author of *Adventures in Minecraft*, a Raspberry Pi trainer, and blogger at stuffaboutco.de

YOU'LL NEED

- ◆ Raspberry Pi wheeled robot
- ◆ Two motor/wheel encoders

There is more to making a robot go in a straight line than just turning the motors on full power – in this tutorial you'll learn how to add encoders to your robot and implement a PID controller to regulate the power.

Anyone who has ever built a wheeled robot will know that driving in a straight line is a lot more difficult than you first think. Sure, holding a true heading for 1, 2 or maybe 3 metres is possible, but keeping it up past 10 or 20 metres without a veer to the left or right becomes astonishingly tricky.

There are many reasons why this happens – uneven surfaces, differences in wheel size, bent axles and, most significantly, the fact that no two motors turn at the same speed! Minor differences in manufacturing and materials result in minor differences in output, and as a result, one motor will spin more quickly than the other. This difference may well be very small, but over time (or distance), it will show as your robot beginning to veer. If the right motor is moving quicker, your robot is going to turn in an arc to the left, and vice versa.

To counter this problem, a solution is required that can accurately measure how fast each motor is moving

and then use this feedback to adjust the motor's speed at run-time so that each motor spins at the same rate.

Encoders are typically used to measure motor speed; these devices provide an output (or pulse) multiple times per revolution.

A PID (proportional-integral-derivative) controller is then used to continuously monitor and adjust motor speed to keep them in sync.

This tutorial steps through adding encoders to a Raspberry Pi-powered robot, using Python to create a PID controller, tuning it to work with your robot, and using the GPIO Zero (gpiozero.readthedocs.io) library to interact with the hardware.

ENCODERS

Encoders come in all shapes, sizes and accuracy. They can be incorporated into motors themselves or as add-ons that connect to the motor shaft or the wheel, but fundamentally they all work in the same way – a consistent signal is provided as the motor turns; the faster the motor is turning, the faster the signal.

A typical robot setup includes a motor controller (or maybe a dedicated HAT), two motors, and a battery pack. In addition, you will need an encoder per motor connected to your Raspberry Pi.

Most encoders will have three or four pins (power, ground, and one or two signal pins); typically the power and ground pins will be connected to a 3.3V and a ground (GND) pin on your Pi; one of the signal pins should be connected to a spare GPIO pin. It's important to check the specifications of your encoders before connecting them up to the Raspberry Pi.

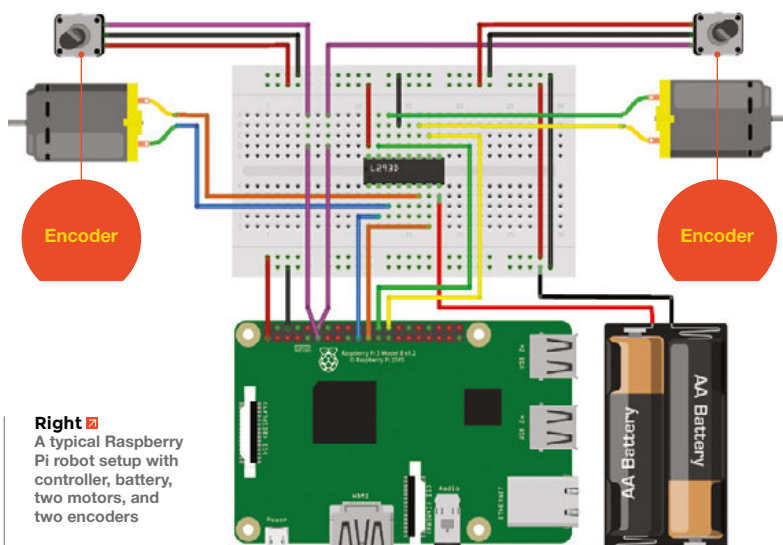
Any Python IDE will do

1. Open up a Python 3 editor (e.g. Thonny) and create a new program.

2. Import the required Python modules:

```
from gpiozero import Robot, DigitalInputDevice
from time import sleep
```

3. Create a constant for sample time – this is how often (in seconds) your program will read the values



Right A typical Raspberry Pi robot setup with controller, battery, two motors, and two encoders

from the encoders – it’s likely that you will need to change this value later to get the best result from your setup:

```
SAMPLETIME = 1
```

4. Create an **Encoder** class to monitor your encoders; this will increment a value each time the pin turns on and off.

```
class Encoder(object):
    def __init__(self, pin):
        self._value = 0

        encoder = DigitalInputDevice(pin)
        encoder.when_activated = self._increment
        encoder.when_deactivated = self._increment

    def reset(self):
        self._value = 0

    def _increment(self):
        self._value += 1

@property
def value(self):
    return self._value
```

5. Use the **gpiozero Robot** class to connect to your motor hardware; each motor will connect to two GPIO pins (one forward, one back), specified as **((left_forward, left_backward), (right_forward, right_backward))** – our robot uses the pins **((10,9), (8,7))**:

```
r = Robot((10,9), (8,7))
```

6. Create two **Encoder** objects passing the GPIO pin the signal connects too; we’ve used GPIO pins 17 and 18:

```
e1 = Encoder(17)
e2 = Encoder(18)
```

7. Start the robot by making the value of both motors 1.0 (forward at full speed):

```
m1_speed = 1.0
m2_speed = 1.0
r.value = (m1_speed, m2_speed)
```

8. Start an infinite loop and print the encoder values:

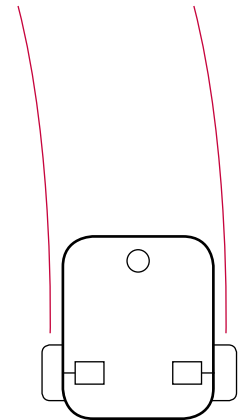
```
while True:
    print("e1 {} e2 {}".format(e1.value, e2.value))
    sleep(SAMPLETIME)
```

9. Run the program.

View the complete **encoder.py** code listing at github.com/martinohanlon/RobotPID.

The **SAMPLETIME** value should be changed to reflect your hardware; you need to find a balance between reading it frequently enough to get good results and slow enough to capture sufficient encoder ticks – try values between 0.1 and 1.0 seconds and aim to capture more than 20 ticks per sample.

Make a note of approximately how many encoder ticks per sample your robot makes.



Above ♦ As the right motor spins quicker than the left, the robot always turns left

PID CONTROLLER

A PID controller continuously calculates an error and applies a corrective action to resolve the error; in this case, the error is the motor spinning at the wrong speed and the corrective action is changing the power to the motor. It is this continuous testing of the motor’s speed and adjusting it to the correct speed which will make your robot’s motors spin at the correct speed and go straight.

PID is a ‘control loop feedback’ mechanism

The controller will have a target motor speed that it wishes to maintain; each time the encoder values are sampled, it will calculate the difference (or error) →

QUICK TIP

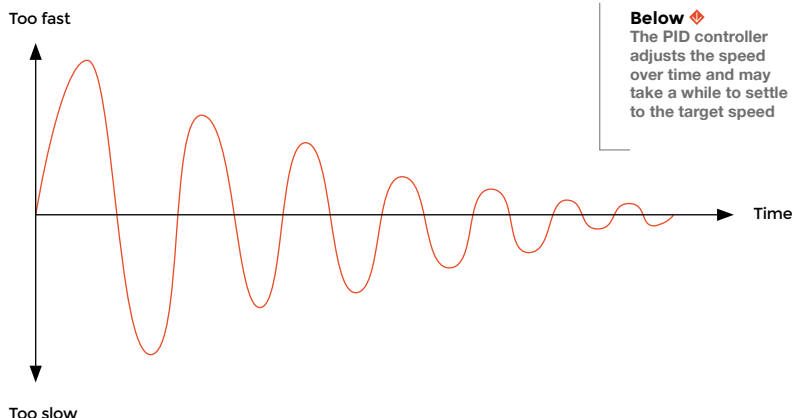
Motors with built-in encoders tend to be more expensive, but they also have much greater accuracy and precision than add-ons.

Below ♦ The PID controller adjusts the speed over time and may take a while to settle to the target speed


TWO-PIN ENCODERS

Your robot maybe fitted with ‘quadrature’ encoders; these encoders use two pins, significantly increase the resolution, and allow the direction the motor is spinning to be determined.

This tutorial assumes you are using simple one-pin pulse encoders, but there is a code example at github.com/martinohanlon/RobotPID which should allow you to modify it. There’s also an excellent write-up at robotoid.com/appnotes/circuits-quad-encoding.html which explains how they work and how to interpret the signals from them.



```
pi@pizerow:~/RobotPID $ python3 encoder.py
e1 0 e2 0
e1 53 e2 54
e1 110 e2 111
e1 166 e2 168
e1 221 e2 223
e1 276 e2 279
e1 330 e2 333
e1 384 e2 387
e1 438 e2 442
e1 490 e2 495
e1 544 e2 550
e1 597 e2 603
e1 653 e2 658
e1 708 e2 713
e1 764 e2 768
e1 819 e2 825
e1 874 e2 880
e1 929 e2 935
e1 983 e2 990
e1 1036 e2 1044
```

Right  Our encoders tick about 50 to 60 times per sample and motor 2 runs slightly faster than motor 1

OTHER PID USES

The input and outputs of a PID controller don't have to be an encoder and a motor; the controller can be applied to any situation where something needs to be constantly monitored and adjusted. This could be:

- Using a magnetometer to make a robot move in a certain direction
- Keeping a camera on a powered mount pointing at the same place
- Making a robot follow a wall by measuring the distance to it with ultrasonic sensors

PID controllers are universal devices and the rules can be applied to solve many different problems.

between the target speed and the actual speed and apply an adjustment to the motor speed. If the adjustment overshoots the next time the encoders are sampled, a smaller opposite adjustment will be made. Over time, the adjustments will even out and the motors will run at a constant speed (or at least that's the theory!).

You will be changing the program you created to read encoder values to calculate an error and apply an adjustment using proportional, derivative, and integral control.

PROPORTIONAL

Proportional control is adjusting the motor speed by adding the value of the error – the value of the error (the difference in encoder ticks between the target and the actual speed) will need to be converted to the motor speed (a value between 0 and 1) by multiplying a constant (KP) to get a 'proportional' change:

$$\text{adjustment} = \text{error} \times \text{KP}$$

Time for maths

Modify the program you created earlier to read encoder values:

1. Add a constant for the target of encoder ticks you want the motors to achieve; make this value about 75% of the 'encoder ticks per sample' value you made a note of earlier (in our case $60 \times 0.75 = 45$):

```
TARGET = 45
```

2. Add a constant (KP) for the proportional change which will be multiplied by the error to create the motor adjustment. This constant will need tuning, but a good starting point is 1 divided by the 'encoder ticks per sample' (e.g. $1 / 60 = 0.0166\text{--}$)

```
KP = 0.02
```

3. At the start of the infinite loop, calculate the error for each motor by subtracting the encoder value from the target:

```
while True:
    e1_error = TARGET - e1.value
    e2_error = TARGET - e2.value
```

4. Calculate the new motor speed by adding the error and multiplying it by the proportional constant:

```
m1_speed += e1_error * KP
m2_speed += e2_error * KP
```

5. The motor speed needs to be between 0 and 1, so clamp the value using `max` and `min`:

```
m1_speed = max(min(1, m1_speed), 0)
m2_speed = max(min(1, m2_speed), 0)
```

6. Update the robot's speed to the new motor values:

```
r.value = (m1_speed, m2_speed)
```

7. Add some debugging code to print the motor speed after the encoder values; this will be useful for tuning:

```
print("e1 {} e2 {}".format(e1.value, e2.value))
print("m1 {} m2 {}".format(m1_speed, m2_speed))
```

8. Before the program sleeps for the sample time, you need to reset the encoders:

```
e1.reset()
e2.reset()
sleep(SAMPLETIME)
```

9. Run your program – you will see the motor's speed being adjusted each time the encoders are sampled, based on the error.

How different are your motors?

View the complete `proportional.py` code listing at github.com/martinohanlon/RobotPID.

Proportional control should be enough to stabilise your motors' speed and keep them turning at about the correct speed, but when there is a large error or you want the speed to adjust quickly, you will get a large overshoot and your robot will react erratically, swinging left to right – this is where derivative control helps.

DERIVATIVE

Derivative control looks at how quickly or slowly the error is changing, creating a larger error if it's changing quickly and a smaller one if slowly. This will help to smooth out the rate of change and prevent erratic changes in speed.

This is achieved by taking the previous error into account when calculating the adjustment and again multiplying by a constant (KD):

$$\text{adjustment} = (\text{error} \times \text{KP}) + (\text{previous_error} \times \text{KD})$$

Modify the program to implement derivative control...

1. Create a new constant (KD) for the derivative control. Again, this value will need to be changed to get the best results for your setup; a good starting value is half the value of KP:

```
KD = 0.01
```

2. Create two variables to hold the previous errors and set them to 0:

```
e1_prev_error = 0
e2_prev_error = 0
```

3. Modify the code which calculate the speeds for motor 1 and 2 to taken into account the previous error:

```
m1_speed += (e1_error * KP) + (e1_prev_error * KD)
m2_speed += (e2_error * KP) + (e1_prev_error * KD)
```

4. At the end of the loop, set the previous error variables to be that of the current error:

```
sleep(SAMPLETIME)
e1_prev_error = e1_error
e2_prev_error = e2_error
```

5. Run your program. Again you will see the motor speed change in relation to error and over time, it should stabilise to a more consistent speed.

Proportional and derivative (PD) control should provide a good level of performance but may not provide consistency of speed over time – integral control can help to bring this stability.

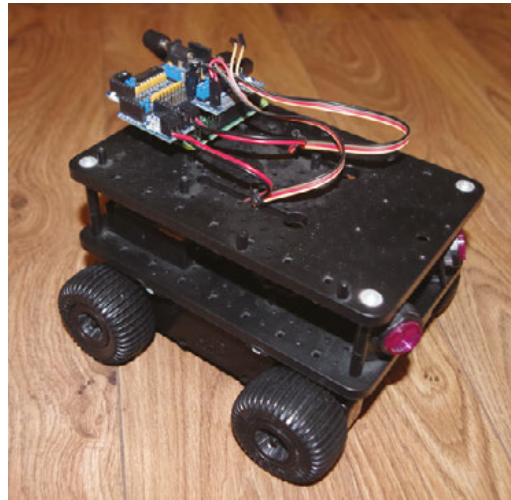
INTEGRAL

Integral control helps to deliver steady state performance by adjusting for slowly changing errors. It does this by keeping a sum of all the previous errors and applying a constant (KI) to the adjustment:

$$\text{adjustment} = (\text{error} \times \text{KP}) + (\text{previous_error} \times \text{KD}) + (\text{sum_of_errors} \times \text{KI})$$

Modify the program to implement integral control:

1. Create a constant for the integral control (KI); a good starting point is half the value of KD:



Left Thanks to a pair of controllers and PID, our robot now runs in a straight line

```
KI = 0.005
```

2. Create two variables to hold the sum of all previous errors and set them to 0:

```
e1_sum_error = 0
e2_sum_error = 0
```

3. Modify the speed calculation to take into account the sum:

```
m1_speed += (e1_error * KP) + (e1_prev_error * KD) + (e1_sum_error * KI)
m2_speed += (e2_error * KP) + (e1_prev_error * KD) + (e2_sum_error * KI)
```

4. At the end of the loop, increment the sum variables by the current error value:

```
sleep(SAMPLETIME)
e1_sum_error += e1_error
e2_sum_error += e2_error
```

5. Run the program. You should see over time that the motor speeds start to stabilise. □

QUICK TIP

You may not have to implement proportional, integral and derivative (PID) control to get your robot to go straight: P or PD might be good enough.

TUNING YOUR SETUP

To get PID control working for your setup, it will need to be tuned; this will involve modifying the constants KP, KD, and KI. There is no exact science to this and there will be a certain amount of trial, error, and intuition required before you find the right setup for your robot.

The following tips however should improve your tuning:

1. Start by modifying the KP constant and get the performance as good as you can before moving onto KD and then finally KI.
2. If the motor adjustments are too aggressive, swinging between too fast and too slow, reduce the constant.
3. If the motor speed isn't changing fast enough, increase the constant.
4. Make any change in small increments; even a very small change can have a dramatic effect.

Once tuned, each motor should settle down to a speed which is close to the encoder target.